CSPP 511-01:
Introduction to Object-Oriented Programming

Harri Hakula

Ryerson 256, tel. 773-702-8584
hhakula@cs.uchicago.edu

July 31, 2000
Outline

- Classes
- User-Defined Types
- Inheritance
- Interface
Classes

In JAVA classes are used to represent concepts. Classes consist of

- methods,
- data members,
- inner classes.

All of these can either be visible only in the scope of the defining class or alternatively in the entire program.
User-Defined Types

User-defined types or classes are different from primitive types. In fact every user-defined type is of at least two type.

Every class is a descendent of the predefined class `Object`. So, every object of every class is of type `Object`, as well as being of the type of its class.

Therefore, every class has some features derived from `Object`. User-defined types are often referred to as derived types.
Inheritance

When we say that a class is a descendent of `Object`, we can think of two natural relationships: either

- a class inherits something from `Object`, or

- a class extends `Object`.

This relationship is implicit and need not be explicitly stated.

However, we can define similar relationships between our own classes. For instance, we can define a base-level implementation which is then further specialized by extending the implementation.
**Example: SSID**

In our SSID example, we defined two methods in our API:

- `public boolean set(String ssid)`
- `public String get()`

A model implementation would be as follows:

```java
public class SSID {
    public boolean set(String ssid) {
        ssid_ = ssid;
        return true;
    }
    public String get() { return ssid_; }
    private String ssid_; }
```
SSID Cntd.

Note that this version is deeply flawed – no attempt to verify that the input is correct is made. We can now define a specialization of SSID that adds verification.

```java
public class SecureSSID extends SSID {
    ... // Constants
    public boolean set(String ssid) {
        if(!verify(ssid)) return false;
        super.set(ssid);
        return true;
    }
    private boolean verify(String ssid) { ... }
}
```
SSID Cntd.

There are two new keywords:

**extends**  This class extends the implementation of SSID. It uses and/or modifies the base class.

**super**  SecureSSID is derived from SSID. SSID is the base class. Every SecureSSID is a SSID. super is the "super class part" of SecureSSID, that is, the SSID part of it.
SSID Cntd.

```java
public boolean set(String ssid) {
    if(!verify(ssid)) return false;
    super.set(ssid);
    return true;
}
```

Note that SecureID has overloaded the method `set`. Through `super.set(ssid)` we can still call the base class implementation.

There is no need to redefine the method `get`, so in absence of it from SecureSSID, the one from SSID is automatically invoked instead.
SSID Cntd.

There are two further points to consider:

- Since SSID has declared the field `ssid_` to be private, SecureSSID cannot access it!
- SecureSSID does not even know how the information is stored in the base class. This SecureSSID would work equally well with String- and integer-based implementations.
IsA vs HasA

We must distinguish between two ways of creating classes:

- inheritance, and
- composition.

These are often called **Is-A** and **Has-A** relationships. If you know that your class "is a" something, use inheritance and specialize, but if your class "has a" something, use composition, i.e., add a field of the type.

A Stack is not a list, but it can have one.
Single Inheritance

Every class can extend only one class explicitly. In some other languages, multiple simultaneous is-A relationships are allowed. In JAVA these classes must be chained even though one could argue that no specialization happens between two base classes.
Tree of Types

All user-defined types form a tree of types, with Object at the root.

It follows that if a method expects a parameter of type Object, every class satisfies this condition! Since JAVA is a statically typed, one must use casts where appropriate. For example:

```java
public int addValues(Object datum) {
    Item item = (Item) datum;
    value += item.value();
}
```

Object does not have method value, but Item does. And, (this is the central argument) every descendant of Item will have it too!
**Interface**

It is now generally accepted that inheritance is not sufficiently easy to use to describe commonly occurring relationships.

Consider the following scenario: The system has two or more items that need to be displayed, e.g., list of items and chunks of free text. Should it be necessary that the list items and the free text are derived from some common base class? What is relevant is that the display mechanism gets the information it needs.

In **JAVA** we can define the interface that the objects have to implement. Interface is shared but not the implementation.
Example: Sorting

Let us return to SelectionSort. The only relationship between two integers that was needed was comparison: $a < b$. We can therefore create an abstraction, an implementation which sorts all arrays of objects that implement the relation $a < b$ or less. The interface is straightforward:

```java
interface Sortable {
    boolean less(Object datum);
}
```

An object which implements a method

```java
boolean less(Object datum)
```

is Sortable.
Sorting Cntd.

Item here is Sortable.

```java
public class Item implements Sortable {
    Item(int value) {
        value_ = value;
    }
    public int value() { return value_; }
    public boolean less(Object datum) {
        Item item = (Item)datum;
        return (value_ < item.value());
    }
    private int value_; // Private member variable for storing the value
}
```
**Sorting Cntd.**

SelectionSort requires only minor modifications:

```java
public class SelectionSort {
    public static void sort(Sortable[] a) { ... }
    private static void interchange(int i, int j, Sortable[] a) {
        Sortable temp;
        ...
    }
    private static int indexOfSmallest(int startIndex, Sortable[] a) {
        Sortable min = a[startIndex];
        ...
```
for (...) 
    if (a[index].less(min)) { ... } 
} 
}

After these modification, an array of Items can be sorted. Of course, integers do not implement our interface and therefore an array of integers cannot be sorted with this algorithm.
Example: Searching

We want to sort so that we can search. Sorting of all Sortable items is easy and need not be written again. But, haven’t we lost our capability to search?

The answer is, no. Why?

\[ a = b \equiv \neg (a < b) \land \neg (b < a) \]

Implementation of binary search for Sortable items is simple:
public class BinarySearch {
    public static int find(Sortable[] a, Sortable value) {
        int lo = 0;
        int hi = a.length - 1;
        while (lo <= hi) {
            int mid = lo + (hi - lo) / 2;
            if (!(a[mid].less(value)) && !(value.less(a[mid])))
                return mid;
            else if (a[mid].less(value)) {
                lo = mid + 1;
            }
        }
    }
}
Again, the original capability is lost, but every sorted array of items implementing the interface `Sortable` can be searched extremely efficiently.